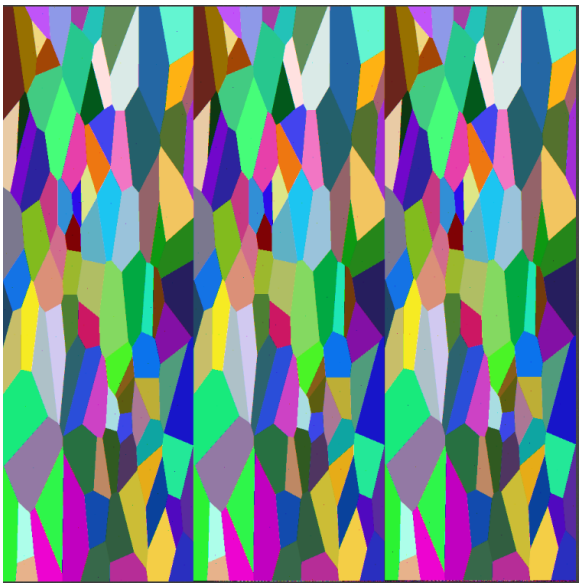
GPU-accelerated Determination of the Voronoi Diagram

CIS667 - FAL19 - Julian Carrasquillo

October 29, 2019



Overview

The main idea is to write a CUDA-based massively multi-threaded application that, when given a set of seeds and a 2-dimensional canvas, computes a discrete approximation of the corresponding Voronoi diagram.

Background

The Voronoi diagram is a fundamental data structure in computational geometry. It has numerous applications in diverse fields ranging from biology to market coverage to cell tower placement. It is defined below:

- Definition: Let S be a set of n sites in Euclidean space of dimension d. For each site p of S, the Voronoi cell V(p) of p is the set of points that are closer to p than to other sites of S.
- The Voronoi diagram V(S) is the space partition induced by Voronoi cells.

In other words, given a set of n seed points, divide the plane into n tiles such that all the points inside a tile are closer to a particular seed p than to any other seed. Clearly, there will be one seed, and only one seed, in every tile.

Computing the Voronoi Diagram

Traditional computational geometry algorithms are applied to subsets of the plane in which real values are used to designate point coordinates. Thus, there are an infinite number of points in the domain of the problem. But practical applications often use rasters, which designate a finite number of points to work with, typically in the form of vector data.

Getting the code in sequence

To more easily handle the multiple values for red, blue, and green levels, in addition to the position of the pixel, I built a simple class object to store all 4 values. Then, an array of the v_point class was made. **Note the image outputs are the array values stitched together 3 times**.

```
class v_point
{
    public:

    long position;
    int red;
    int green;
    int blue;
};
```

The center points were selected from the uniform distribution with range `0 - dim * dim` using the `random` library. In addition to these values, the red, green, and blue values were selected from the uniform distribution with range `0 - 255`.

CUDA migration

The main algorithm in the sequential version of the code used the `v_point` class to compare the `.position` member of a given point to the list of known centers:

```
v_point closest_center(v_point point, v_point center_list[], long center_list_size, long dim){
    float distance;
    float shortest_dist = 0.0;
    long x_point, y_point;
    long x_center, y_center;
    v_point closest_center;

    // extract x and y components for the checked point
    x_point = point.position / dim;
    y_point = point.position % dim;


    // run distances against all listed centers
    for(long i = 0; i < center_list_size; i++){
        // extract x and y components for the center
        x_center = center_list[i].position / dim;
        y_center = center_list[i].position % dim;
        distance = sqrt(pow(1.0 * x_center - x_point, 2) + pow(1.0 * y_center - y_point, 2));

        // are we in the first iteration? Take that distance
        if(i == 0){
            shortest_dist = distance;
            closest_center.position = center_list[i].position;
            closest_center.red = center_list[i].red;
            closest_center.blue = center_list[i].blue;
            closest_center.green = center_list[i].green;

        // if not, then check to see if the new distance we calculated is smaller
        // note this produces a first calculated point for contested areas
        } else if(distance < shortest_dist){
            shortest_dist = distance;
            closest_center.position = center_list[i].position;
            closest_center.red = center_list[i].red;
            closest_center.blue = center_list[i].blue;
            closest_center.green = center_list[i].green;
        }
    }
    return closest_center;
}
```

With the calculations occuring in this part of the code, I looked to adapt it to fit in the CUDA framework. With more time, an attempt at a 2-dimensional handling with the y-versions of the familiar `blockIdx.x` and `threadIDx.x`could be tackled. For simplicity, each thread generated handled the for loop to compare their pixel to each center pixel in the image.

```
__global__ void cu_calc_dist(v_point *pixels_d, v_point *centers_d, long array_size, long centers_size)
{
    long i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    long j = 0;
    long x_point, y_point, x_center, y_center;
    float distance, shortest_dist;
    v_point closest_center;

    if(i < array_size*array_size){
        x_point = pixels_d[i].position / array_size;
        y_point = pixels_d[i].position % array_size;

        for(j = 0; j < centers_size; j++){
            x_center = centers_d[j].position / array_size;
            y_center = centers_d[j].position % array_size;
            distance = sqrt(pow(1.0 * x_center - x_point, 2) + pow(1.0 * y_center - y_point, 2));

            if(j == 0){
                shortest_dist = distance;
                closest_center.position = centers_d[j].position;
                closest_center.red = centers_d[j].red;
                closest_center.blue = centers_d[j].blue;
                closest_center.green = centers_d[j].green;

            // if not, then check to see if the new distance we calculated is smaller
            // note this produces a first calculated point for contested areas
            } else if(distance < shortest_dist){
                shortest_dist = distance;
                closest_center.position = centers_d[j].position;
                closest_center.red = centers_d[j].red;
                closest_center.blue = centers_d[j].blue;
                closest_center.green = centers_d[j].green;
            }
        }
        pixels_d[i] = closest_center;
    }
}
```

The CUDA code behaved pretty well with the new class type. I expected some pushback with respect to using `__host__` or `__device__` key words, but there was no need for it. Setting up pointers to your new class and allocating the correct amount of space suffices.

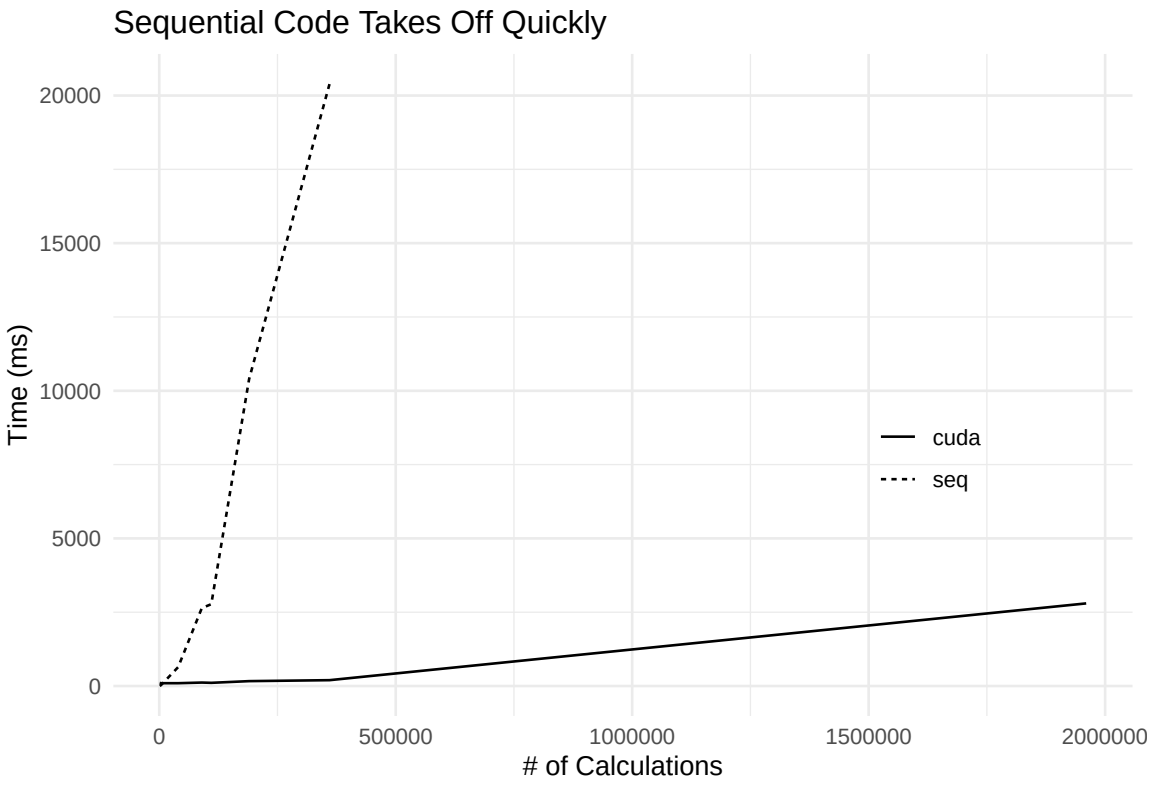
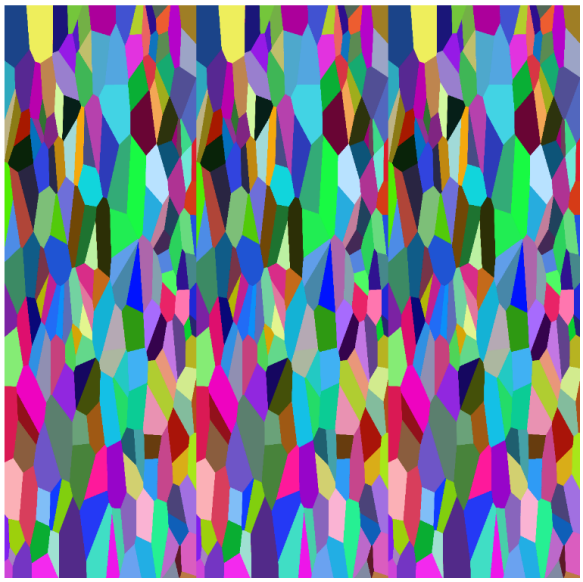


Image Dim	# of Centers	# of Calculations	Time, Seq	Time, CUDA	Speed Up
100	10	900	5	107	0
200	20	3600	30	96	0
500	100	40000	642	97	7
1000	100	90000	2642	118	22
750	200	110000	2772	109	25
2000	100	190000	10397	166	63
2000	200	360000	20388	199	103
10000	200	1960000	NA	2799	NA

The amount of time taken to complete even a 2000 pixel image with 200 centers was showing how slowly the sequential code was running. The CUDA code was used for 1,000,000 pixels and 200 centers, completing the process in 2,800 milliseconds.



Discussion

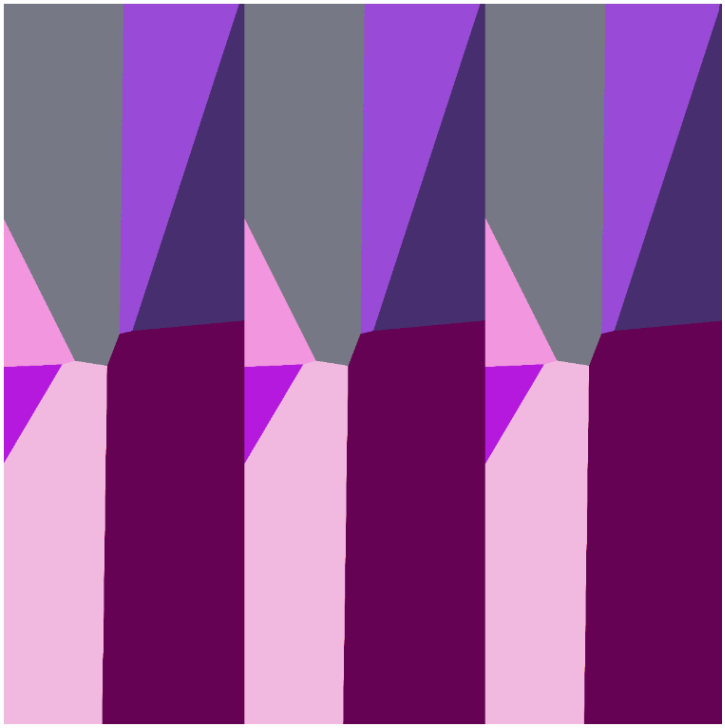
The process saw a very obvious upgrade in performance with a relatively basic approach. There is room to introduce 2 dimensional processing by considering the matrix of dimensions dim^2 x # of Centers. With this introduction, a sort of atomic minimum needs to be implemented. As it stands, each thread knows the smallest distance calculated because it owns the process.

In scenarios where the centers are relatively fixed, like cell phone towers, natural landmarks, or retail locations for example, the array of centers could be passed as a set of constants, potentially avoiding the need to allocate memory each time.

For a more realworld solution, Google Maps data can be incorporated to develop distances in terms of driving time. How far is point x to point y, assuming you have to get across traffic in LA, for example. There could be some interesting city planning implementations.

Flipping the Algorithm?

To explore, I reversed the algorithm to find the farthest point and assume its color. With 200 centers, only 7 colors remained.



Appendix

Kernal Function

```
__global__ void cu_calc_dist(v_point *pixels_d, v_point *centers_d, long array_size, long centers_size)
{
    long i = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    long j = 0;
    long x_point, y_point, x_center, y_center;
    float distance, shortest_dist;
    v_point closest_center;

    if(i < array_size*array_size){
        x_point = pixels_d[i].position / array_size;
        y_point = pixels_d[i].position % array_size;

        for(j = 0; j < centers_size; j++){
            x_center = centers_d[j].position / array_size;
            y_center = centers_d[j].position % array_size;
            distance = sqrt(pow(1.0 * x_center - x_point, 2) + pow(1.0 * y_center - y_point, 2));

            if(j == 0){
                shortest_dist = distance;
                closest_center.position = centers_d[j].position;
                closest_center.red = centers_d[j].red;
                closest_center.blue = centers_d[j].blue;
                closest_center.green = centers_d[j].green;

            // if not, then check to see if the new distance we calculated is smaller
            // note this produces a first calculated point for contested areas
            } else if(distance < shortest_dist){
                shortest_dist = distance;
                closest_center.position = centers_d[j].position;
                closest_center.red = centers_d[j].red;
                closest_center.blue = centers_d[j].blue;
                closest_center.green = centers_d[j].green;
            }
        }

        pixels_d[i] = closest_center;

    }
}
```

Caller

```
// This function is called from the host computer.
// It manages memory and calls the function that is executed on the GPU
extern void calc_distance(v_point *pixels, v_point *centers, long array_size, long centers_size)
{
    // build GPU counterpart for each class array on host
    v_point *pixels_d;
    v_point *centers_d;
    cudaError_t result;

    // allocate space in the device
    result = cudaMalloc ((void**) &pixels_d, sizeof(v_point) * array_size * array_size);
    if (result != cudaSuccess) {
        fprintf(stderr, "cudaMalloc - 'pixels' failed.");
```

```
        exit(1);
    }

    result = cudaMalloc ((void**) &centers_d, sizeof(v_point) * centers_size);
    if (result != cudaSuccess) {
        fprintf(stderr, "cudaMalloc - 'centers' failed.");
        exit(1);
    }

    //copy the array from host to *_d in the device
    result = cudaMemcpy (pixels_d, pixels, sizeof(v_point) * array_size * array_size, cudaMemcpyHostToDevice);
    if (result != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy - 'pixels' failed.");
        exit(1);
    }

    result = cudaMemcpy (centers_d, centers, sizeof(v_point) * centers_size, cudaMemcpyHostToDevice);
    if (result != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy - 'centers' failed.");
        exit(1);
    }

    // set execution configuration
    dim3 dimblock (BLOCK_SIZE);
    dim3 dimgrid (ceil((float) array_size*array_size/BLOCK_SIZE));

    // actual computation: Call the kernel
    cu_calc_dist <<<dimgrid, dimblock>>> (pixels_d, centers_d, array_size, centers_size);

    // transfer results back to host
    result = cudaMemcpy (pixels, pixels_d, sizeof(v_point) * array_size * array_size, cudaMemcpyDeviceToHost);
    if (result != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy copy to host failed. %s\n", cudaGetErrorString(result));
        exit(1);
        cudaFree(pixels_d);
        cudaFree(centers_d);
    }

    // release the memory on the GPU
    cudaFree(pixels_d);
    cudaFree(centers_d);
}
```